B3M33HRO HW5
Grasping - Comments

1 Common Mistakes

- **Inverse sorting of ϵ quality** - the best grasps have ϵ-quality of 1 and the worst have -1. So, you should have sorted the quality from 1 to -1.
- **Not processing the point cloud enough for GPD** - you have been told to make the GPD run as fast as possible. The best way to achieve this is to downsample the point cloud. Some of you had used point cloud with 200 000 points, or even more, resulting in run time of 5 second and more.

2 Comments - No points were taken for this

The comments in this section are not tightly connected to this course and there will be no questions about it in the exam. However, we think it may be useful to you in the future.

- **Concatenation of the point clouds** - most of you correctly used the overloaded "+" operation to concatenate the point cloud. However, some students tried to to concatenate the point clouds "by hand". There is nothing bad about it, but it should be done effectively. Numpy enables you to work in vectorized way, almost like in Matlab. So, if possible, do not use excessive for-loops to iterate over the items of an array and rather use logical indexing.

  Also, do not use Numpy concatenation functions (vstack, hstack, ...) to append items one by one. It will copy all the points to a new array almost every time, which is super slow. If you really need to append, use the default python list which is created as a pointer (and optionally create Numpy array from it, if needed). The difference for 100 000 three-dimensional points (which is not so much) can be seen in Figure 1.

```
In [1]: import numpy as np
        import time

In [2]: data = np.random.randn(100000, 3)

        start_python = time.time()
        data_python = []
        for d in data:
            data_python.append(d)
        data_python = np.array(data_python)

        print(f"Python list took {time.time()-start_python} seconds.")

        start_numpy = time.time()
        data_numpy = np.empty((0, 3))
        for d in data:
            data_numpy = np.vstack((data_numpy, d))

        print(f"Numpy concatenation took {time.time()-start_numpy} seconds.")

        Python list took 0.11512279510498047 seconds.
        Numpy concatenation took 39.521745443344116 seconds.

In [3]: print(f"Shape of the array created with list is {data_python.shape}")
        print(f"Shape of the array created with vstack is {data_numpy.shape}")
        print(f"The two list are equal: {np.all(data_python == data_numpy)}")

        Shape of the array created with list is (100000, 3)
        Shape of the array created with vstack is (100000, 3)
        The two list are equal: True
```

Figure 1: Comparison of appending to an array with Python list and Numpy vstack.

- **Sorting** - even though, there is a lot of ways how to sort and array in Python, the basic functions *sort()* or *sorted()* are written in C and are super effective. In addition, you can easily insert Lambda function in the *key* argument to effectively sort the arrays. The grasps in the assignment could have been easily sorted like this:

  ```
  grasps.sort(key=lambda x: x.epsilon_quality, reverse=True)
  ```

  Or, if you want to rather create a copy of the object:

  ```
  grasps_sorted = sorted(grasps, key=lambda x: x.epsilon_quality, reverse=True)
  ```

- **Outliers removal** - some of you used the Statistical or Radius outliers removals. You need to be careful with these functions as they require to tune the parameters and require knowledge about basic statistics. And not only in Open3D, because these functions are the basic ones implemented in many libraries, *e.g.,* PCL library. For example, the Statistical removal works on rejecting points which are more than user-defined number of standard deviations σ away from the neighbors. You want to look at bigger neighborhood—looking at 20 point near themselves will do nothing (this applies also for the radius removal). And then reject points with distances bigger than 2 or 3 standard deviations (Three-sigma rule). Some of you used small neighborhood and low number of σ (under 0.1). It rejected a lot of points, but not in the correct way.

- **Downsampling** - you usually do not need to have point cloud with millions of points. Working with it is much more resource demanding and almost all functions which will work with the point cloud will take a lot of time to complete—normals estimation, mesh creation, ICP, *etc.* And also when concatenating point clouds from multiple cameras, more points does not mean better resolution (which is also not needed in most cases), but some of the point will be doubled with really small difference in position, as the calibration of the cameras will never be perfect. For example, using voxel-based downsampling will give you more sparse (and "quick") point cloud and will also have the effect of removing these doubled points. On the other hand, you do not want the point cloud to be too sparse, as some important (usually geometrical) features can be lost.

  And even though we talk only about point clouds this can be also applied to other types of data.